Idee: Ist es möglich,

```
for (int i = 1; i < 10; ++i) {
    // ...
}
```

zu

```
for (auto i: range(1, 10)) {
    // ...
}
```

zu vereinfachen? Prinzipiell ja: Wir benötigen eine entsprechende Klasse mit einem zugehörigen Iterator-Typ, der die Operatoren $*$ und $++$ unterstützt.

range.hpp

```
template <typename T>
class IntegralRange {
   public:
      IntegralRange(T begin_val, T end_val) :
         begin_val(begin_val), end_val(end_val) {
      }

      class Iterator {
         // ...
      };

      Iterator begin() const {
         return Iterator(begin_val);
      }
      Iterator end() const {
         return Iterator(end_val);
      }

   private:
      T begin_val;
      T end_val;
};
```

range.hpp

```
class Iterator {
   public:
      Iterator(T val) : val(val) {
      }
      T operator*() {
         return val;
      }
      Iterator& operator++() {
         ++val; return *this;
      }
      Iterator& operator++(int) {
         Iterator it = *this;
         ++val; return it;
      }
      bool operator==(const Iterator& other) const {
         return val == other.val;
      }
      bool operator!=(const Iterator& other) const {
         return val != other.val;
      }

   private:
      T val;
};
```

Das müsste nun in eine entsprechende Template-Funktion verpackt werden, die den Typparameter automatisch bestimmt:

```
template<typename T>
IntegralRange<T> range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Problem: Diese Template-Funktion akzeptiert zunächst prinzipiell beliebige T, geht dann aber schief, wenn T sich nicht wie ein ganzzahliger Datentyp verhält. Ist es möglich, hier eine Einschränkung vorzunehmen?

- Wenn der C++-Übersetzer alle in Frage kommenden Kandidaten einer Template-Funktion in Betracht zieht, werden zunächst die Template-Parameter nur in die Template-Parameter-Deklaration und die Funktionsdeklaration (Parameter und Return-Typ) eingesetzt und danach überprüft, ob das Resultat semantisch zulässig ist.
- Wenn die Antwort nein ist, dann ist das kein Fehler. Stattdessen wird nur ganz einfach der Kandidat aus dem Pool der Kandidaten entfernt.
- Das Prinzip nennt sich SFINAE: *Substitution Failure Is Not An Error*.

Angenommen, wir wollen bei *range* nur **int** und **unsigned int** zulassen. Dann könnten wir das so organisieren:

```
template <typename T> struct is_integer {};
template <> struct is_integer<int> {
    using type = IntegralRange<int>;
}
template <> struct is_integer<unsigned int> {
    using type = IntegralRange<unsigned int>;
}

template<typename T>
typename is_integer<T>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Hier wird nun *is_integer*<*T*>::*type* benutzt, das nur für die Fälle **int** und **unsigned int** definiert ist. Für alle anderen Typen gibt es keinen *type*, was zu einem Ausschluss per SFINAE führt.

Es lohnt sich, die Bedingungen (wie hier *is_integer*) von dem gewünschten Typ zu trennen (war hier *IntegralRange*<*T*>). Die Standard-Bibliothek bietet hierfür *std*::*enable_if*, das so definiert sein könnte:

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T; };
```

Der erste Parameter ist die **bool**-Bedingung, der zweite Parameter spezifiziert den gewünschten Typ.

Als nächstes wird eine Hilfsklasse benötigt, die auf der Template-Ebene einen konstanten Wert eines integralen Typs repräsentiert. Die C++-Bibliothek bietet hierfür *std*::*integral_constant* an, das wie folgt implementiert sein könnte:

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant;
    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

Darauf basierend lassen sich Hilfsklassen für **bool**-Werte definieren:

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

Nun lässt sich die Aufzählung der zugelassenen Typen unabhängig vom Resultat-Typ umsetzen:

```
template <typename T> struct is_integer :
    public std::false_type{};
template <> struct is_integer<int> :
    public std::true_type{};
template <> struct is_integer<unsigned int> :
    public std::true_type{};

template<typename T>
typename std::enable_if<
    is_integer<T>::value, // boolean-valued condition
    IntegralRange<T> // wanted type, if correct
>::type // is the wanted type, if correct
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

In <*type_traits*> finden sich aber bereits eine Vielzahl einzelner Tests, u.a. auch *std*::*is_integral*, der alle integralen Typen umfasst:

```
template<typename T>
typename std::enable_if<
    std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Wenn wir eine weitere Variante für Zeiger und Iteratoren zulassen wollen, könnten wir im einfachsten Falle die ganzzahligen Typen ausschließen:

```
template<typename T>
typename std::enable_if<
    !std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

Es wäre aber besser, die auf *IntegralRange* basierende Variante auf Typen zu beschränken, die wie Zeiger oder Iteratoren aussehen, d.h. die die unären Operatoren $*$ und $++$ unterstützen.
Wir benötigen hierzu etwas Handwerkszeug, um per SFINAE die Unterstützung von Operatoren und Methoden zu testen:

- Mit *std*::*declval* können wir aus einem Typ ein Objekt des Typs machen, ohne zu wissen, wie der Konstruktor aussieht.
- Mit **decltype** kann der Typ eines Ausdrucks wie ein Typname verwendet werden.
- Mit *std*::*remove_reference* werden wir && bzw. & los.

std::*declval* ist eine Template-Funktion aus <*utility*>, die nur deklariert, jedoch nie definiert wird:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval();
```

- Wir dürfen *declval* somit nur in Konstruktionen einsetzen, die vollständig zur Übersetzzeit ausgewertet werden und bei denen nur der Typ relevant ist, jedoch nicht die (nicht vorhandene) Definition der Funktion.
- Mit *add_rvalue_reference* bleiben Referenztypen so wie sie sind, nur Typen ohne Referenz (also weder & noch &&) werden mit && versehen.
- Diese Ergänzung ist notwendig, um beispielsweise die Existenz von Methoden testen zu können, die nur für *rvalue*-Objekte zugänglich sind: **struct** *foo* { **void** *bar*()&& { /*... */} };

Angenommen, wir haben einen Template-Parameter für einen Zeigertyp oder Iterator und wollen den Typ nach der Dereferenzierung haben:

```
template<typename T>
struct dereferenced {
    using type = decltype(*std::declval<T>());
};
```

- Innerhalb von **decltype** wird nur der Typ eines Ausdrucks bestimmt, dieser jedoch nicht bewertet. Deswegen ist *std*::*declval*<*T*>() zulässig und liefert ein virtuelles Objekt des Typs *T*, das wir dann dereferenzieren können, um davon mit **decltype** den Typ zu bestimmen.
- *dereferenced*<**int***>::*type* und *dereferenced*<*std*::*vector*<**int**>::*iterator*> entsprechen nun jeweils **int**.

Nun lässt sich das so kombinieren, dass wir abprüfen, ob die Operatoren $*$ und $++$ unterstützt werden:

```
template<typename T>
typename std::remove_reference<
    decltype(
        *std::declval<T&>(), // * supported?
        ++std::declval<T&>(), // ++ supported?
        /* wanted type, comes with &&: */
        std::declval<IteratorRange<T>>()
    )
>::type // now with && removed
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

Da diese Lösungen wenig elegant aussehen und auch nicht entgegenkommend sind bei Fehlermeldungen, wurde bereits seit langer Zeit in C++ über ein geeigneteres Sprachmittel nachgedacht, den sogenannten *concepts*:

- Bislang sind die *concepts* noch nicht in den Standard (zuletzt C++17) aufgenommen worden. Die Aufnahme in C++20 ist aber geplant.
- Es gibt eine technische Spezifikation für *concepts* (ISO/IEC TS 19217:2015), die kurz als *concepts TS* bezeichnet werden. Diese Variante wird vom g++ mit der Option „-fconcepts“ unterstützt. Ein dem nahekommender Draft steht unter N4549 zur Verfügung.
- Für C++20 wurden Überarbeitungen vorgeschlagen (P0587R0), die zu der in P0734R0 dokumentierten Arbeitsfassung für C++20 geführt haben.
- Die C++20-Fassung ist bislang nirgends umgesetzt. Die folgenden Beispiele lassen sich mit „-fconcepts“ übersetzen.

Die wichtigste Ergänzung sind die ⟨requires-expression⟩s, die es erlauben, Anforderungen zu spezifizieren:

⟨primary-expression⟩ ⟶ ⟨requires-expression⟩

⟨requires-expression⟩ ⟶ **requires**
[⟨requirement-parameter-list⟩]
⟨requirement-body⟩

⟨requirement-parameter-list⟩ ⟶ „(“
[⟨parameter-declaration-clause⟩] „)“

⟨requirement-body⟩ ⟶ „{“ ⟨requirement-seq⟩ „}“

⟨requirement-seq⟩ ⟶ ⟨requirement⟩

⟶ ⟨requirement-seq⟩ ⟨requirement⟩

⟨requirement⟩ ⟶ ⟨simple-requirement⟩

⟶ ⟨type-requirement⟩

⟶ ⟨compound-requirement⟩

⟶ ⟨nested-requirement⟩

Einzelne Anforderungen innerhalb eines ⟨requirement-body⟩ können wie folgt aussehen:

⟨simple-requirement⟩ ⟶ ⟨expression⟩ „;“

⟨type-requirement⟩ ⟶ **typename** [⟨nested-name-specifier⟩] ⟨type-name⟩ „;“

⟨compound-requirement⟩ ⟶ „{“ ⟨expression⟩ „}“ [**noexcept**] [⟨trailing-return-type⟩] „;“

⟨nested-requirement⟩ ⟶ ⟨requires-clause⟩ „;“

Zu den Deklarationen werden die Konzepte hinzugefügt, die anschließend in Typspezifikationen referenziert werden können:

$$
\begin{aligned}
\langle\text{decl-specifier}\rangle &\longrightarrow \textbf{concept} \\
\langle\text{simple-type-specifier}\rangle &\longrightarrow \langle\text{constrained-type-specifier}\rangle \\
\langle\text{constrained-type-specifier}\rangle &\longrightarrow \langle\text{qualified-concept-name}\rangle
\end{aligned}
$$

forward-iterator.hpp

```
template<typename T>
concept bool ForwardIterator = requires(T it1, T it2) {
   { it1 == it2 } -> bool;
   { it1 = it2 } -> T&;
   { *it1 };
   { ++it1 } -> T;
};
```

- *ForwardIterator* ist ein Beispiel für ein *concept*, das die wichtigsten Anforderungen eines solchen Iterators aufführt.
- Hier sind die Anforderungen erfüllt wenn die einzelnen ⟨compound-requirement⟩s zulässig sind und, sofern angegeben, den entsprechenden Datentyp zurückliefern.
- Bei Concepts TS muss da noch jeweils **bool** als Datentyp für das Konzept angegeben werden, bei C++20 wird dies wohl entfallen, weil in diesem Kontext nur **bool** sinnvoll ist.

Concepts TS: Anforderungen bei Template-Deklarationen

Bei den Template-Deklarationen kommt dann die Möglichkeit hinzu, Anforderungen an die Template-Parameter hinzuzufügen:

⟨template-declaration⟩ ⟶ **template** „<“ ⟨template-parameter-list⟩ „>“
[⟨requires-clause⟩] ⟨declaration⟩

⟨requires-clause⟩ ⟶ **requires** ⟨constraint-expression⟩

⟨constraint-expression⟩ ⟶ ⟨logical-or-expression⟩

range.hpp

```
template <typename T>
requires std::is_integral<T>::value
class IntegralRange {
   // ...
};
```

- Anforderungen können jetzt sehr leicht auch in Template-Deklarationen integriert werden.
- Und bei Template-Funktionen muss nicht mehr auf SFINAE zurückgegriffen werden:

range.hpp

```
template<typename T>
requires std::is_integral<T>::value
IntegralRange<T> range(T begin_val, T end_val) {
   return IntegralRange<T>(begin_val, end_val);
}
```

range.hpp

```
template <typename IT>
requires ForwardIterator<IT>
class IteratorRange {
   // ...
};
```

- Anforderungen können sich auf benannte Konzepte beziehen, sowohl bei Template-Klassen als auch bei Template-Funktionen.

range.hpp

```
template<typename T>
requires ForwardIterator<T>
IteratorRange<T> range(T begin_it, T end_it) {
   return IteratorRange<T>(begin_it, end_it);
}
```

- Traits sind Charakteristiken, die mit Typen assoziiert werden.
- Die Charakteristiken selbst können durch Klassen repräsentiert werden und die Assoziationen können implizit mit Hilfe von Templates oder explizit mit Template-Parametern erfolgen.
- Über <*type_traits*> können diverse Eigenschaften von Typen abgefragt oder getestet werden.

Sum.hpp

```
#ifndef SUM_HPP
#define SUM_HPP

template <typename T>
inline T sum(const T* begin, const T* end) {
   T result = T();
   for (const T* it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Die Template-Funktion *sum* erhält einen Zeiger auf den Anfang und das Ende eines Arrays und liefert die Summe aller enthaltenen Elemente.
- (Dies ließe sich auch mit Iteratoren lösen, darauf wird hier jedoch der Einfachheit halber verzichtet.)

Summe eines Arrays

TestSum.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = " << sum(text, text + dim(text)) <<
      std::endl;
}
```

- In den beiden Tests mit **int** und **float** klappt des problemlos, jedoch nicht mit **char**...
- Bei den ersten beiden Arrays funktioniert das Template recht gut. Weswegen scheitert es im dritten Fall?

```
thales$ testsum
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = ,
thales$
```

- Wieso wird „,“ ausgegeben, wenn wir eine numerische Summe erwartet hätten?

Definition einer Traits-Template-Klasse

SumTraits.hpp

```
#ifndef SUM_TRAITS_HPP
#define SUM_TRAITS_HPP

// by default, we use the very same type
template <typename T>
class SumTraits {
   public:
      using SumValue = T;
};

// special case for char
template <>
class SumTraits<char> {
   public:
      using SumValue = int;
};
#endif
```

- Die Template-Klasse *SumTraits* liefert als Charakteristik den jeweils geeigneten Datentyp für eine Summe von Werten des Typs T.
- Per Voreinstellung ist das T selbst, aber es können Ausnahmen definiert werden wie hier zum Beispiel für *char*.

Sum2.hpp

```
#ifndef SUM2_HPP
#define SUM2_HPP

#include "SumTraits.hpp"

template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin,
      const T* end) {
   using SumValue = typename SumTraits<T>::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Statt *T* wird hier jetzt *SumTraits*<*T*>::*SumValue* als Typ für die Summe verwendet.

TestSum2.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum2.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = "
      << sum(text, text + dim(text)) << std::endl;
}
```

```
thales$ testsum2
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = 3372
thales$
```

Sum3.hpp

```
#ifndef SUM3_HPP
#define SUM3_HPP

#include "SumTraits.hpp"

template <typename T, typename ST = SumTraits<T>>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- C++ unterstützt voreingestellte Template-Parameter, seit C++11 auch bei Template-Funktionen.
- Diese Konstruktion ermöglicht dann einem Nutzer dieser Konstruktion die Voreinstellung zu übernehmen oder bei Bedarf eine eigene Traits-Klasse zu spezifizieren.

TestSum3.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum3.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

int main() {
   int numbers[] = {2147483647, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   std::cout << "sum of numbers[] = " <<
      sum<int, MyTraits>(numbers, numbers + dim(numbers)) << std::endl;
}
```

```
thales$ testsum3
sum of numbers[] = -2147483639
sum of numbers[] = 2.14748e+09
thales$
```

Sum4.hpp

```
template <typename ST, typename T>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}
template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin,
      const T* end) {
   return sum<SumTraits<T>, T>(begin, end);
}
```

- Den automatisierbar bestimmbaren Template-Parametern sollten diejenigen vorangehen, die u.U. abweichend bestimmt werden.
- Umgekehrt gilt, dass den Template-Parameter mit Voreinstellungen nicht solche ohne Voreinstellungen folgen dürfen.
- Der Konflikt lässt sich durch zwei Varianten lösen: einer generellen mit zwei Template-Parametern und dem Spezialfall mit nur einem Template-Parameter.

TestSum4.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum4.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

int main() {
   int numbers[] = {2147483647, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   std::cout << "sum of numbers[] = " <<
      sum<MyTraits>(numbers, numbers + dim(numbers)) << std::endl;
}
```

- Nun muss nur noch die Traits-Klasse angegeben werden, jedoch nicht mehr der Elementtyp des Arrays, der sich aus dem Parameter ableiten lässt.

- Statischer Polymorphismus basiert in C++ auf Templates und der Verlagerung der semantischen Überprüfung auf den Zeitpunkt der Instantiierung.
- Ein wesentliches Element ist die automatisierte Auswahl der am besten passenden Template-Klasse oder Template-Funktion zur Übersetzzeit.
- Die elegante Erweiterbarkeit ergibt sich aus der Möglichkeit, dass Varianten einfach per **#include** hinzugefügt werden können. Dann werden sie implizit überall berücksichtigt, wo sie anwendbar sind.
- Traits erlauben es, Eigenschaften von Typen zur Übersetzzeit zu spezifizieren und auszuwerten.
- Sowohl bei Template-Klassen als auch bei Template-Funktionen sind per SFINAE frei definierbare Einschränkungen möglich. Bei generischen Klassen ohne Einschränkungen besteht die Gefahr, dass die elegante Erweiterbarkeit für weitere Spezialfälle nicht mehr möglich ist.
- Mit *concepts* wird dies hoffentlich bald sehr viel eleganter möglich sein.